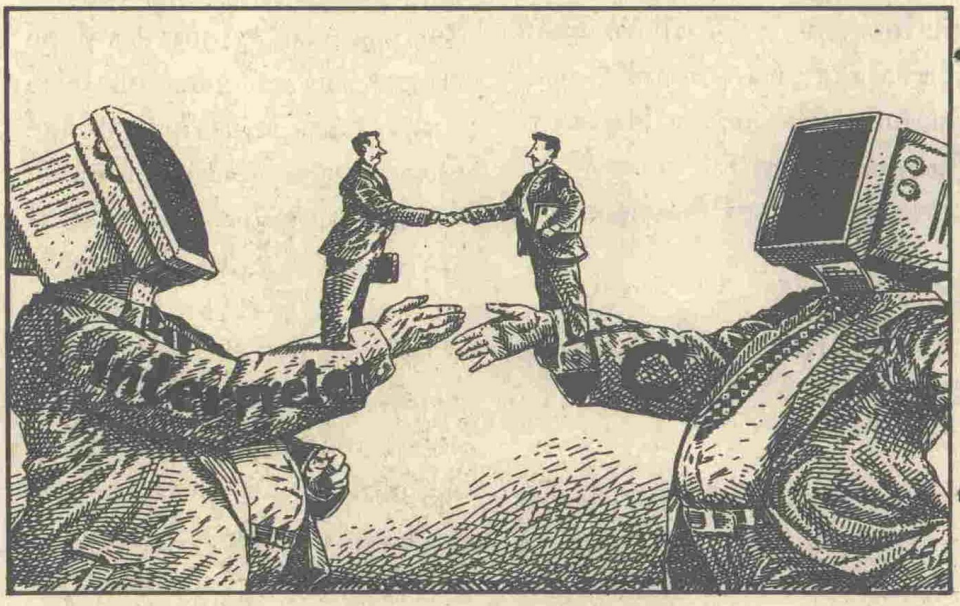
# เขียน C Interpreter ด้วยซี (1)

*ไม่ต้องเสียเวลา*
*คอมพายเลอร์ซีให้ยุ่งยาก*
*อินเตอร์พรีตเตอร์ช่วยคุณได้*




คุณสมบัติอย่างหนึ่งของภาษาซี คือ การที่ตัวมันเองเป็นภาษาที่สามารถนำไปใช้เขียนโปรแกรมเพื่อสั่งให้เครื่องคอมพิวเตอร์ทำงานประเภทไหนก็ได้ (ที่เครื่องมีขีดความสามารถทำได้) อย่างที่เรียกกันว่าภาษาใช้งานทั่ว ๆ ไป แต่ที่สำคัญยิ่งกว่า คือความสามารถที่จะใช้เขียนโปรแกรมประเภทจัดการระบบ (system programming) ซึ่งเป็นผลงานที่สร้างชื่อเสียงให้กับภาษาซีอย่างมาก เห็นจะได้แก่การถูกนำเอามาเขียนระบบการทำงานของยูนิกซ์ที่ได้กลายเป็นโอเอส (operating system) ที่ได้รับความนิยมอย่างมากในเครื่องรุ่นใหญ่หรือแม้แต่เครื่องไมโครคอมพิวเตอร์รุ่นใหม่ ๆ ที่ใช้โปรเซสเซอร์ที่มีความสามารถสูง เช่น 80386 หรือ 80486 ก็มีการนำเอาโอเอสตัวนี้มาใช้เช่นกัน และก็คงไม่ใช่เหตุบังเอิญที่คนหันมานิยมใช้ภาษาซีกันเพราะว่ายูนิกซ์เกิดดังขึ้นมา ซึ่งโดยเหตุผลที่แท้จริงนั้นคือซีเป็นภาษาโครงสร้างที่เขียนและอ่านได้ง่ายเหมือนภาษาระดับสูงหลาย ๆ ภาษา เช่น อัลกอลหรือปาสคาล ในขณะที่มีคุณสมบัติหรือขีดความสามารถทำงานได้เร็วและสั่งระบบให้ทำงานได้ใกล้เคียงกับภาษาระดับต่ำ เช่น แอสเซมบลี ฉะนั้นจึงไม่น่าแปลกใจเลยที่ภาษาซีเป็นภาษาที่มีคอมพายเลอร์ที่สามารถทำงานได้บนเครื่องคอมพิวเตอร์ทุกรุ่นทุกระดับ ไม่ว่าจะใหญ่หรือเล็ก และยิ่งไปกว่านั้นในเครื่องแต่ละชนิดก็ยังมีคอมพายเลอร์ภาษาซีให้เลือกใช้ได้อีกหลายตัว เช่น เครื่องไมโครคอมพิวเตอร์ไอบีเอ็มและเครื่องเลียนแบบ (compatible) ทั้งหลายที่หลาย ๆ คนใช้กันอยู่นี้ก็มีคอมพายเลอร์ภาษาซีให้ใช้ได้อย่างเหลือเฟือไม่ว่าจะเป็น Microsoft C, Lattice C, Quick C หรือ Turbo C ซึ่งสิ่งที่อยากจะเน้นคือภาษาซีเป็นภาษาที่คุ้มค่ากับการเรียนรู้จริง ๆ

## สร้าง C Interpreter ด้วยคอมพายเลอร์ซี

โอกาสนี้เราก็จะลองดูวิธีการใช้ภาษาซีเพื่อเขียนตัวแปลภาษาซี (interpreter) ซึ่งก็จะมีวิธีการทำงานในแบบโต้ตอบกับผู้ใช้ทันที (เป็นวิธีการที่เหมือนกับที่ใช้อยู่กับภาษาเบสิกหรือในโปรแกรมดีเบสที่เรารู้จักกันดี) แทนที่จะต้องนำเอาโปรแกรมที่เขียนขึ้นไปทำการแปล (compile) ด้วยคอมพายเลอร์ก่อนเหมือนกับตัวภาษาซีเอง อย่างไรก็ตามตัวแปลภาษาซีที่เราจะเขียนกันดูนี้จะเป็นเพียงส่วนย่อย (subset) ของภาษาซี (ANSI C) ที่จะมีวิธีการทำงานของภาษาเหมือนกัน แต่ถ้าต้องการจะเพิ่มขีดความสามารถ

(ซึ่งไม่มีอยู่ในตัวแปลภาษาตัวนี้) ก็สามารถทำได้ตัวแปลภาษาแบบอินเตอร์พรีตเตอร์นี้ ก็คงจะอยู่ที่คุณสมบัติในด้านความเป็นกันเองกับผู้ใช้ คือมีการโต้ตอบกันทันทีทำให้ผู้เขียนรู้ว่าคำสั่ง (statement) ที่เขียนขึ้นนั้นถูกต้องหรือไม่ และถ้าผิดก็สามารถที่จะทำการแก้ไขได้ทันทีและด้วยคุณสมบัตินี้เอง เราอาจใช้วิธีการทำงานของอินเตอร์พรีตเตอร์มาใช้งานกับฐานข้อมูล (database) เพื่อทำหน้าที่เรียกใช้ข้อมูลในแบบ query language ได้อย่างดี ฉะนั้นพอจะเป็นที่คาดหวังไว้ได้เลยว่าอินเตอร์พรีตเตอร์ที่เราสร้างกันดูนี้จะอำนวยความสะดวกในการเขียนภาษาซีอีกเยอะ โปรแกรมนี้จะเรียกว่า 'Little C'

## เค้าโครงของ Little C

ภาษาซีเป็นภาษาคอมพิวเตอร์ที่จัดว่ามีขีดความสามารถสูงมาก ทั้ง ๆ ที่คีย์เวิร์ดของภาษาที่กำหนดไว้โดย ANSI นั้นมีอยู่เพียง 32 ตัวเท่านั้น (bulit-in commands) ฉะนั้นเราจะมาสร้างกันแค่ 'Little C' ซึ่งเป็นชุดย่อยของซีก็พอ สำหรับ 'Little C' นี้ก็จะรวมเอาชุดคำสั่งเด่น ๆ หรือส่วนที่มีความจำเป็นเบื้องต้นในการเขียนภาษาซีเอาไว้ทั้งหมด ไม่ว่าจะเป็นส่วนของฟังก์ชัน recursive หรือตัวแปรโกลบอลหรือตัวแปลโลคอล นอกจากนี้ก็รวมถึงความสามารถในการสร้างลูปและการใช้คำสั่ง if แต่ไม่มีการกำหนดให้สามารถใช้งาน 'switch statement'

### คุณสมบัติ Little C

ทีนี้ก็มาดูกันว่า Little C Interpreter มีคุณสมบัติอย่างไรกันบ้าง เพื่อที่จะทำให้รู้ว่ามีอะไรบ้างที่สามารถทำได้และมีอะไรบ้างที่ทำไม่ได้ ส่วนคุณสมบัติเบื้องต้นที่กำหนดไว้จะมีดังนี้

- สามารถสร้างฟังก์ชันต่าง ๆ ที่มีการเรียกใช้ตัวพารามิเตอร์และตัวแปรแบบโลคอลได้
- มีขีดความสามารถในการสร้างและเรียกใช้งานโปรแกรมแบบเรียกใช้ตัวเอง (recursion)
- มีคุณสมบัติในการทำงานแบบ 'if-statement'
- สามารถเขียนโปรแกรมและทำงานในแบบวนลูปโดยใช้คำสั่ง 'do-while', 'while' และ 'for' ได้
- สามารถสร้างและเรียกใช้ตัวแปรแบบเลขจำนวนเต็ม (integer) และอักขระ (character veriables) ได้
- กำหนดให้มีตัวแปรแบบข้อความ (string constants) ได้ แต่มีขีดจำกัดในการใช้งานอยู่บ้าง
- สามารถที่จะใช้คำสั่ง 'return statement' ได้ทั้งแบบที่มีการส่งค่าคืนและไม่มีการส่งค่าใด ๆ คืนมาด้วย
- มีการกำหนดฟังก์ชัน (standard library functions) ไว้เป็นจำนวนจำกัด
- เครื่องหมาย (operators) ที่มีให้ใช้ประกอบด้วย +, −, *, /, %, <, >, <=, >=, ==, :=, unery −, และ unery +
- มีฟังก์ชันที่มีการส่งคืนค่าเป็นเลขจำนวนเต็ม
- สามารถใส่ข้อความเพื่ออธิบายโปรแกรม (comments) ได้

ทั้งหมดนี้คือคุณสมบัติหรือขีดความสามารถของ Little C interpreter ข้อจำกัดที่สำคัญอย่างหนึ่งของ Little C คือการเขียนคำสั่งของคำสั่ง 'if', 'while', 'do' และ 'for' จะต้องอยู่ในขอบเขตของเครื่องหมายวงเล็บปีกกา (curly braces) ซึ่งเราไม่สามารถจะใช้วิธีการเขียนลอย ๆ (โดยไม่มีเครื่องหมายวงเล็บปีกกา) เหมือนที่ทำได้ในภาษาซีตามปกติ ฉะนั้นถ้าเราต้องการจะเขียนโปรแกรมดังรูปที่ 1

เราสามารถที่จะทำได้ถ้าเขียนด้วยภาษาซีตามแบบมาตรฐาน แต่ Little C จะถือว่าผิดกฎเกณฑ์ ฉะนั้นเราจะต้องเขียนโปรแกรมนี้ใหม่ดังรูปที่ 2

```
for (a=0; a>10; a=b+1
   for (b=0:b<10; b=b+1)
      for (c=0, c<10; c=c+1)
         put ("hi");
if (.....)
      if (.....) statement;
```

รูปที่ 1

```
for (a=0; a<10: a=a+1) {
   for (b=0: b<10; b=b+1) {
      for (c=0; c<10: c=c+1) {
         puts ("hi");
      }
   }
}
if (.....) {
   if (.....) {
      statement;
   }
}
```

รูปที่ 2

จะเห็นว่าในโปรแกรมที่แก้ไขใหม่นี้ จะมีการใช้เครื่องหมายวงเล็บปีกกาเพื่อกำหนดขอบเขตของชุดคำสั่งทุกช่วง ถึงแม้ว่าจะถือเป็นข้อจำกัดของ 'Little C' แต่ในการเขียนโปรแกรมด้วยภาษาซีจริง ๆ นั้นก็ควรที่จะใช้วิธีการนี้เช่นกัน เพราะนอกจากจะทำให้โปรแกรมดูเป็นระเบียบแล้วยังเป็นการป้องกันความสับสนในการทำความเข้าใจกับการทำงานของโปรแกรมต้นฉบับ (Source code) ด้วย อย่างไรก็ตามถ้าต้องการที่จะแก้ไขให้ Little C สามารถใช้ได้แบบเดียวกับซีมาตรฐาน (ไม่ต้องใช้เครื่องหมายวงเล็บปีกกา) ก็สามารถแก้ไขได้

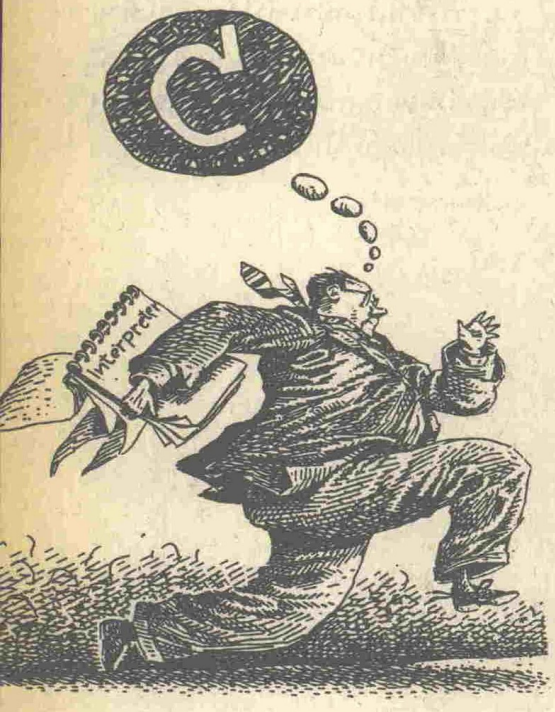
## การเขียนอินเตอร์พรีตเตอร์กับภาษาโครงสร้าง

ตัวแปลภาษาที่เราคุ้นเคยกันมากที่สุดในบรรดาภาษาคอมพิวเตอร์ทั้งหลายก็คงเป็นภาษาเบสิกอินเตอร์พรีตเตอร์ ซึ่งผู้ใช้เครื่องไมโครคอมพิวเตอร์ทั้งหลายใช้มาโดยตลอด ภาษาเบสิกนี้เป็นภาษาที่ไม่เป็นโครงสร้างและได้ถูกออกแบบมาเพื่อให้เป็นอินเตอร์พรีตเตอร์ให้กับภาษาที่ไม่มีโครงสร้างจึงดูเหมือนจะทำได้ง่ายกว่าภาษาที่เป็นโครงสร้าง เช่น ภาษาซีและจุดที่ทำให้การเขียนอินเตอร์-พรีตเตอร์ทำได้ยากกว่าคือการใช้ตัวแปร ซึ่งในภาษาเบสิกนั้นตัวแปรทั้งหมดจะมีลักษณะเป็นโกลบอล (เป็นตัวแปรที่สามารถเรียกใช้ได้จากทุก ๆ ส่วนของโปรแกรม) ในขณะที่ภาษาประเภทโครงสร้างอย่างซีนั้นมีการใช้ตัวแปรทั้งแบบโกลบอลและโลคอล (โลคอลคือตัวแปรที่จะเรียกใช้ได้เฉพาะส่วนของโปรแกรม) รวมทั้งยังมีโปรแกรมย่อยที่แยกต่างหากจากโปรแกรมไว้ให้ใช้ได้ด้วย

อย่างไรก็ตามในกรณีที่ทำการเขียนตัวแปลเพื่อให้สามารถใช้งานฟังก์ชันหรือตัวแปรแบบโลคอลและการเรียกใช้ตัวเองไว้เรียบร้อยแล้วนั้นภาษาซีจะสามารถแปลได้ง่ายกว่าภาษาเบสิก เพราะภาษาเบสิกนั้นมีข้อยกเว้นอยู่มากมายในระดับของทฤษฎี เช่น ในภาษาเบสิกนั้นเครื่องหมาย '=' (equal sign) สามารถเป็นได้ทั้งเครื่องหมายกำหนดค่า (assignment operator) ในคำสั่งและยังทำหน้าที่เป็นเครื่องหมายแสดงความเท่ากัน (equality operator) ใน (relational statement) ด้วย ซึ่งภาษาซีจะมีการใช้เครื่องหมายต่าง ๆ ซ้ำกัน (inconsistancies) ในหลายความหมายน้อยมาก ทำให้ช่วยลดความยุ่งยากในการตีความได้อย่างมาก

ตามที่เราได้กล่าวไปแล้วว่าภาษาซีเป็นภาษาโครงสร้าง ซึ่งยอมให้มีการสร้างและใช้งานโปรแกรมย่อยที่แยกกันเอาไว้ต่างหากได้ โดยที่โปรแกรมย่อยเหล่านี้สามารถจะมีตัวแปรของตัวเองแยกเป็นอิสระได้ด้วย นอกจากนี้ยังยอมให้มีการเรียกใช้ตัวเองได้อีกและคุณสมบัติต่าง ๆ เหล่านี้ก็มีส่วนที่ทำให้การเขียนตัวแปลทำได้ยากกว่าการเขียนตัวแปลภาษาที่ไม่เป็นโครงสร้างหรือแม้แต่การเขียนตัวแปลแบบคอมพายเลอร์เอง เช่น เมื่อคอมพายเลอร์ทำการสร้างรหัสเพื่อให้มีการเรียกใช้ฟังก์ชันนั้นมันก็เพียงแต่เก็บ (push) ตัวอาร์กิวเมนต์ (argument) ทั้งหลายที่มีเข้าไปไว้ในสแต็กของระบบ (system stack) แล้วจึงทำการประมวลผลคำสั่ง 'CALL'' ไปที่ฟังก์ชันที่ต้องการ และเมื่อทำงานตามหน้าที่ของฟังก์ชันที่เรียกใช้เสร็จแล้วก็จะคืนค่าที่ได้มาไว้ในตัว accumulator ของซีพียูแล้วทำการยกเลิกการใช้งานสแต็กและประมวลผลคำสั่ง 'RET'

แต่ถ้าเป็นอินเตอร์พรีตเตอร์แล้ววิธีการทำงานก็ต้องเริ่มด้วยการเรียกใช้



ฟังก์ชันซึ่งจะต้องทำการยกเลิกงานที่กำลังทำอยู่ทั้งหมด โดยทำการเก็บค่าของสถานะต่าง ๆ (current state) ในขณะนั้นเอาไว้ทั้งหมดแล้วจึงไปหาฟังก์ชันที่เรียกใช้ ต่อจากนั้นจะทำการประมวลผลฟังก์ชันและเก็บค่าของผลลัพธ์ที่ได้ก่อนที่จะกลับมาที่จุดซึ่งเรียกใช้ฟังก์ชันนั้นแล้วจึงคืนค่าทั้งหมดที่เก็บเอาไว้และทำงานต่อไป ฉะนั้นจึงมีความจำเป็นที่จะต้องสร้างรหัสที่สามารถทำงานได้เหมือนกับคำสั่ง 'CALL'' และ 'RETURN' หรือแม้แต่การยอมให้สามารถเรียกใช้ตัวเองได้ด้วยนั้น การสร้างตัวแปลแบบคอมพายเลอร์ก็สามารถทำได้ง่ายกว่าการสร้างอินเตอร์พรีตเตอร์ด้วยเช่นกัน

## โครงสร้างพื้นฐานของภาษาซี

ก่อนที่เราจะสร้างตัวแปลตามที่ได้พูดกันไว้ควรจะทำความรู้จักกับโครงสร้างพื้นฐานของภาษาซีกันก่อน เพื่อให้เห็นและเข้าใจถึงเค้าโครงของภาษานี้ ซึ่งจะเป็นเพียงบางส่วนของภาษาซีตามมาตรฐาน ANSI และถ้าใครต้องการที่จะรู้กับเรื่องนี้เพิ่มเติมสามารถหาอ่านได้จากเอกสารต่าง ๆ ที่พูดถึงข้อกำหนดของภาษาซีตามมาตรฐานที่กำหนดใน ANSI

หลักการเบื้องต้นของภาษาซีคือโปรแกรมที่เขียนด้วยภาษาซีนี้จะมีลักษณะเป็นการเรียกใช้ฟังก์ชันทั้งสิ้น โดยจะมีการใช้ตัวแปรร่วมถ้าจำเป็น ส่วนตัวฟังก์ชันเองนั้นก็จะประกอบด้วยชื่อฟังก์ชันและตัวพารามิเตอร์ต่าง ๆ ที่ต้องการใช้รวมถึงชุดคำสั่งต่าง ๆ (block of code) ที่จะทำงานตามหน้าที่ของฟังก์ชันนั้น ๆ โดยที่ชุดคำสั่งเหล่านี้สามารถกำหนดขอบเขตได้ด้วยการใช้เครื่องหมายวงเล็บปีกกาโดยเริ่มด้วยเครื่องหมายวงเล็บเปิด ('{') ในส่วนของคำสั่งสามารถที่จะเริ่มต้นด้วยคีย์เวิร์ดของภาษาซีเอง เช่น 'if' หรือเป็นนิพจน์ได้และจากหลักเกณฑ์พื้นฐานนี้สามารถกำหนดกฎเกณฑ์ดังรูปที่ 3

```
program    ---> collection of functions (plus global variables)
function   ---> function-spacifier parameter-list code-block
code-block ---> {statement sequence}
statement  ---> keyword; expression; or code-block
```

รูปที่ 3

```
count = 100; /* line 1 /
sample = i/22 *(c-10) /* line 2 */
printf ("this is an expression"); /* line 3 */
```

รูปที่ 4 แสดงชุดคำสั่งในภาษาซี

และที่สำคัญอีกอย่างหนึ่งคือ โปรแกรมภาษาซีนั้นจะเริ่มต้นประมวลผลที่ฟังก์ชัน 'main ( )' เสมอ และจะสิ้นสุดลงเมื่อพบเครื่องหมาย '}' หรือคำสั่งรีเทอร์นตัวสุดท้ายที่พบในฟังก์ชัน 'main ( )' โดยมีข้อยกเว้นว่าภายในฟังก์ชันนี้จะต้องไม่มีการเรียกใช้ฟังก์ชันบางชุดที่มีผลให้ยกเลิกการประมวลผลโปรแกรม เช่น 'exit ( )' หรือ 'abort ( )' ส่วนฟังก์ชันอื่น ๆ ที่มีการเรียกใช้ระหว่างการประมวลผลฟังก์ชัน 'main ( )' ก็สามารถจะทำได้ไม่ว่าจะเป็นการเรียกใช้ทางตรงหรือทางอ้อม ซึ่งหมายความว่าโปรแกรมจะเริ่มทำงานที่จุดเริ่มต้นของฟังก์ชัน 'main ( )' แล้วจบลงเมื่อประมวลผลฟังก์ชันนี้เสร็จและ Little C ก็จะทำงานในลักษณะที่ว่านี้ด้วยเช่นกัน

### นิพจน์ในภาษาซี

คำสั่งในภาษาซีจะเป็นได้ทั้งคำสั่งหลัก เช่น 'while' หรือ 'switch' ฉะนั้นลักษณะของชุดคำสั่งที่จะแสดงให้ดูต่อไปนี้จะถือว่าเป็นนิพจน์ในภาษาซีทั้งหมดดังรูปที่ 4

จะเห็นได้ว่านิพจน์ของภาษาซีนั้นอาจจะแปลกไปกว่าภาษาอื่น ๆ เช่น ในตัวอย่างบรรทัดแรก (/* line 1 */) นั้น เครื่องหมาย '=' จะถือเป็นเครื่องหมายในภาษาซี ซึ่งจะถูกกำหนดให้ทำหน้าที่ไม่เหมือนกับที่ใช้ในภาษาอื่น เช่น เบสิก เพราะในภาษาเบสิกนั้นค่าหรือผลลัพธ์ที่ได้จากทางด้านขวาของ '=' จะถูกกำหนดให้กับตัวแปรที่อยู่ทางซ้ายของเครื่องหมาย ส่วนในภาษาซีนั้นจะมีความหมายกว้างกว่านี้เพราะ '=' หมายถึง เครื่องหมายซึ่งจะมีผลให้ค่าหรือผลลัพธ์ใด ๆ ที่อยู่ทางขวาของเครื่องหมายจะมีค่าเท่ากับตัวแปรที่อยู่ทางซ้ายเสมอคือจะทำงานในลักษณะของการเป็นนิพจน์ด้วย จากคุณสมบัติข้อนี้จะทำให้ภาษาซีสามารถเขียนนิพจน์ของคำสั่งในลักษณะดังต่อไปนี้ได้คือ

```
a=b=c=100:
printf ("%d", a=4+5);
```

เหตุที่สามารถเขียนในลักษณะนี้ได้เพราะเป็นการหาค่านอกเหนือจากธรรมดา

สำหรับตัวอย่างในบรรทัดที่ 2 (/* line 2 */) เป็นการกำหนดค่าที่มีความซับซ้อนมากขึ้น ส่วนในตัวอย่างที่ 3 (/* line 3 */) ฟังก์ชัน 'printf ( )' จะถูกเรียกเพื่อให้แสดงผลลัพธ์เป็นข้อความ ซึ่งโดยปกติแล้วฟังก์ชันของภาษาซีจะต้องทำการคืนค่าให้กับโปรแกรมไม่ว่าจะมีการกำหนดหรือไม่ก็ตาม ฉะนั้นฟังก์ชันต่าง ๆ จะถือว่าเป็นนิพจน์ที่ต้องคืนค่าใด ๆ ให้ด้วย ถึงแม้ว่าจะมีการกำหนดค่าที่ได้ให้กับอะไรก็ตาม ANSI ได้กำหนดให้สามารถใช้คำสั่ง 'void' เพื่อยอมให้ฟังก์ชันที่ถูกเรียกใช้โดยไม่จำเป็นต้องคืนค่ามาให้ก็ได้ แต่ใน Little C ของเราจะไม่มีคุณสมบัติข้อนี้

โครงสร้างของนิพจน์ในภาษาซีจะมีรูปแบบคล้ายคลึงกับภาษาอื่น ๆ คือจะมีการเรียกใช้ตัวเองได้ ส่วนในเรื่องของเครื่องหมายนั้น 'Little C' ของเราก็จะมีเครื่องหมายต่าง ๆ ให้ใช้ตามนี้ คือ +, −, *, /, %, =, เครื่องหมายเปรียบเทียบ (relational operator) เช่น <,==,> เป็นต้น รวมทั้งเครื่องหมายวงเล็บ (parentheses) ด้วย โดยมีกฎเกณฑ์ (preduction rules) ของเราจะเป็นรูปที่ 5

ในที่นี้ rel-op หมายถึงเครื่องหมายเปรียบเทียบต่าง ๆ ของภาษาซี ส่วน lvelue และ rvalue หมายถึงสิ่งต่าง ๆ ที่จะอยู่ทางซ้ายและขวาของเครื่องหมายประโยคคำสั่งตามลำดับ อย่างไรก็ตามมีอยู่สิ่งหนึ่งที่จะต้องระวังคือการกำหนดลำดับความสำคัญของเครื่องหมายต่าง ๆ ที่จะถูกกำหนดเอาไว้ในกฎเกณฑ์ด้วย

```
expression  ----> [assignment] [rvalue]
assignment  ----> lvalue = rvalue
lvalue      ----> veriable
rvalue      ----> part [rel-op part]
part        ----> term [+term] [ ] term]
term        ----> factor [* factor] [/factor] [% factor]
factor      ----> part [rel-op part]
part        ----> [+ or −] atom
atom        ----> variable constant, function, or (expression)
```

รูปที่ 5 แสดงกฎเกณฑ์ต่าง ๆ ในภาษาซี

โดยเครื่องหมายที่อยู่ทางด้านล่างจะมีลำดับสูงกว่าเครื่องหมายที่อยู่ทางด้านบน

การจัดลำดับความสำคัญของเครื่องหมายต่าง ๆ เอาไว้เพื่อเป็นแนวทางให้โปรแกรมทำการประมวลผลนิพจน์ที่มีลำดับของเครื่องหมายสูงกว่า ฉะนั้นการประมวลผลนิพจน์จะเป็นไปตามกฎเกณฑ์นี้เสมอ เช่น

```
count = 10 −5 *3;
```

ซึ่งจะถูกประมวลผลตามกฎเกณฑ์ข้อที่ 1 ก่อน โดยจะแยกนิพจน์ออกเป็น 3 ส่วน คือ

```
count = 10−5*3
lvalue assignment operator rvalue
```

และเมื่อพบว่าไม่มีการใช้เครื่องหมายเปรียบเทียบภายใน 'rvalue' ซึ่งเป็นส่วนย่อยของนิพจน์ก็จะทำการเรียกใช้กฎในส่วนของ 'term' ซึ่งก็จะทำให้ได้ผลลัพธ์ดังนี้

```
10          −          5*3
term        minus      term
```

สิ่งที่ต้องทำต่อไปคือการพิจารณาในส่วนของ 'term' ตัวที่สองประกอบไปด้วยค่า (factor) 2 ตัว คือ 5 และ 3 ซึ่งต่างก็เป็นค่าคงที่ด้วยกันทั้งคู่และเป็นระดับต่ำสุดของกฎที่ไม่สามารถจะแตกย่อยออกไปได้อีก ฉะนั้นขั้นต่อไปคือการประมวลผลนิพจน์โดยจะเริ่มด้วยการคูณ 5 กับ 3 (5×3) จะได้ผลลัพธ์เป็น 15 แล้วจึงนำเอาไปลบออกจาก 10 จะได้ผลลัพธ์ในขั้นสุดท้ายเป็น −5 ที่จะถูกนำเอาไปเป็นค่าของตัวแปร count ซึ่งวิธีการที่เราได้พูดถึงกันมาทั้งหมดนี้จะเป็นหลักการที่นำมาใช้ในการเขียน 'Little C' ด้วย

## เริ่มกันที่ Expression Parser

ตัวแปลภาษาที่จะทำหน้าที่ในการแยกแยะและตรวจสอบนิพจน์ที่เขียนขึ้นว่าถูกต้องและเป็นไปตามกฎเกณฑ์ของภาษาหรือไม่นั้น คือ expression parser ซึ่งถือได้ว่าเป็นส่วนสำคัญที่สุดส่วนหนึ่งของตัวแปลภาษาเลยก็ย่อมได้ เพราะภาษาซีนั้นมีการออกแบบนิพจน์ที่มีคุณสมบัติกว้างขวางกว่าภาษาอื่น ๆ ซึ่งจะเห็นได้ว่าคำสั่งต่าง ๆ ที่เขียนขึ้นด้วยวิธีการของภาษาซีนั้นต้องผ่านการตรวจสอบจาก expression parser ด้วยกันทั้งสิ้น

ส่วนวิธีการที่จะนำมาใช้ในการออกแบบ expression parser มีอยู่ด้วยกันหลายแบบ แต่ที่นิยมให้กันในบรรดาคอมพายเลอร์ที่เขียนขึ้นเพื่อการค้าจะใช้เทคนิคแบบ table-driven parser ซึ่งโดยปกติแล้วก็จะสร้างขึ้นมาด้วย parser-generator program เพราะวิธีการแบบนี้

มีข้อดีที่สามารถทำงานได้เร็วกว่าวิธีการอื่น ๆ แต่ก็มีข้อเสียหรือข้อยุ่งยากกว่าวิธีอื่น ๆ สำหรับ Little C Interpreter ของเรานั้นจะใช้วิธีการแบบ recursive-descent parser ซึ่งจะใช้หลักของเหตุและผล (logic) ที่กำหนดให้กับกฎเกณฑ์

ในส่วนของ recursive-descent parser เองจะประกอบไปด้วยฟังก์ชัน recursive เป็นจำนวนหลาย ๆ ฟังก์ชัน ที่ทำหน้าที่ตามกระบวนการของนิพจน์ตามที่ตรวจพบซึ่งถ้าเป็น parser ที่ใช้อยู่ในคอมพายเลอร์แล้วจะทำการสร้างรหัส (object code) ที่เหมาะสมให้กับคำสั่งที่ตรวจพบ แต่สำหรับตัวแปลของเรานั้นจะทำหน้าที่ในการประมวลผลนิพจน์ที่กำหนดไว้นั้นเลย ส่วนขีดความสามารถของตัวแปลหรือ Little C จะเป็นไปตามที่เราได้เคยพูดเอาไว้ในตอนต้น

การทำงานของ parser โดยทั่วไปแล้วตัวแปลภาษาแบบอินเตอร์พรีตเตอร์ทั้งหลาย (หรือแม้แต่คอมพายเลอร์เอง) จะทำหน้าที่พิเศษที่จะคอยอ่านชุดคำสั่งและแปลงให้อยู่ในรูปแบบที่สามารถจะทำงานในขั้นตอนต่อไปได้ ซึ่งวิธีการที่ทำกันมานานแล้วคือการแยกคำสั่งต่าง ๆ นั้นออกเป็นประเภท ๆ และเรียกว่า 'tokens' โดยภาษาคอมพิวเตอร์โดยทั่ว ๆ ไป (รวมทั้งภาษาซีด้วย) จะให้ความหมายของโปรแกรมในลักษณะของ tokens นี้เช่นกัน ฉะนั้น token นี้ก็จะได้แก่ส่วนย่อยที่สุดของโปรแกรมที่ไม่สามารถจะแยกย่อยออกไปได้อีก ตัวอย่างเช่น เครื่องหมายเท่ากับ ('==') ก็คือ token ตัวหนึ่งเพราะเครื่องหมายนี้ไม่สามารถแยกออกไปให้เป็นอย่างอื่นได้อีก (โดยไม่ให้ความหมายของเปลี่ยนแปลงไป) ในทำนองเดียวกันคำสั่ง 'if' ก็เป็น token อีกตัวหนึ่งถึงแม้ว่าตัว 'i' และ 'f' ต่างก็สามารถจะนำมาใช้ในความหมายอื่นในภาษาซีได้

สำหรับภาษาซีตามมาตรฐาน ANSI นั้นตัว token ทั้งหลายจะถูกกำหนด (defined) เอาไว้เป็นประเภท ๆ ไป โดยจะมีทั้งหมด 6 ประเภทด้วยกันคือ keywords, identfiers, constants, string, operators และ punctuetion ซึ่ง token แต่ละประเภทจะหมายถึงสิ่งต่าง ๆ ดังนี้

- keywords ได้แก่ token ที่ถูกสร้างเอาไว้ใช้กับภาษาซี โดยจะทำหน้าที่ใดหน้าที่หนึ่งเฉพาะเจาะจงลงไป เช่น 'while', 'if' และ 'for' เป็นต้น
- Identifiers คือตัวแปรต่าง ๆ รวมทั้งฟังก์ชันและสิ่งต่าง ๆ ที่ผู้ใช้กำหนดขึ้นเอง แต่สำหรับ Little C ผู้ใช้จะกำหนดขึ้นเองไม่ได้
- Constants และ strings คือค่าคงที่และข้อความต่าง ๆ ที่ใช้ในโปรแกรม
- Operators หมายถึงเครื่องหมายต่าง ๆ ที่มีใช้ได้ในภาษารวมถึงเครื่องหมายทางคณิตศาสตร์ เครื่องหมายเปรียบเทียบ และเครื่องหมายอื่น ๆ อีก
- Punctuation คือเครื่องหมายหรือสัญลักษณ์ที่ใช้ประกอบในการเขียนโปรแกรม เช่น ',' (comma) และ ';' (semicolon) เป็นต้น ซึ่งเครื่องหมายบางตัวในกลุ่มนี้อาจจะซ้ำกับที่อยู่ในกลุ่มของ operators ขึ้นอยู่กับวิธีการนำเอาไปใช้เพื่อแสดงความหมายของโปรแกรม

## การแยกประเภทของ token parser

การแยกประเภท tokenของ parser จะสมมติให้ทำการแยกชุดคำสั่งดังต่อไปนี้

```
for (x=0, x<10, x=x+1)
    printf ("hello %d",x),
```

ซึ่ง parser จะทำหน้าที่แยกคำสั่งต่าง ๆ เหล่านี้ออกเป็นกลุ่ม ๆ เพื่อให้สามารถทำงานในขั้นตอนต่อไปได้ สำหรับคำสั่งชุดนี้จะถูกแยกออกเป็นประเภทต่าง ๆ ดังรูปที่ 6

เพื่อที่จะใช้ตัวแปล Little C ของเราทำงานได้สะดวกขึ้นจะให้มีการแบ่ง

| Token | Category |
|---|---|
| for | keyword |
| ( | punctuation |
| x | identifier |
| = | operator |
| 0 | constant |
| ; | punctuation |
| x | identifier |
| < | operator |
| 10 | constant |
| ; | punctuation |
| x | identifier |
| + | operator |
| 1 | constant |
| ) | punctuation |
| printf | identifier |
| ( | punctuation |
| "hello %d" | string |
| , | punctuation |
| x | identifier |
| ) | punctuation |
| ; | punctuation |

รูปที่ 6 ชุดคำสั่งใน parser

ประเภทของ token ออกไปดังรูปที่ 7

โดยกำหนดให้ฟังก์ชันที่จะทำหน้าที่แปลงตัว token ต่าง ๆ จากโปรแกรมให้กับ Little C นั้นชื่อว่า 'get__token ( )' สำหรับตัวแปลร่วมและตัวแปรแบบ enumeration type นั้นจะเป็นไปดังรูปที่ 8

| Token typer | Includes |
|---|---|
| delimeters | punctuation และ operators |
| key words | keywords |
| strings | quoted strings |
| identifiers | variable และ function names |
| number | numeric constant |
| block | {หรือ} |

รูปที่ 7 การแบ่งประเภทของ token

```
char * prog: /* points to current location in source code */
extern char  *p__buf: /* points to start of program buffer */
char token [80], /* hold strings representation of token */
char token__type: /* contains the type of token */
char tok: /* holds the internal representation of token if
                        it is a key word */
enum tok__types {DELIMITER, IDENTIFIER, NUMBER,
                KEYWORD, TEMP, STRING, BLOCK};

/* These are the constants used to call entx__err ( )
when a syntax error occurs, Add more if you like,
NOTE: SYNTAX is a generic error message used when nothing
else seems appropriste
*/
enum error__msg
{SYNTAX, UNBAL__PARENS, NO__EXP, EQUALS__EXPERTED,
NOT__VAR, PARAMO__ERR, SEMI__EXPECTED,
UNBAL__BRACES, FUNC__UNDEF, TYPE__EXPECTED,
NEST__EXPECTED, QUOTE__EXPECTED, NOT__TEMP
TOO__MANY__LVARS};
```

รูปที่ 8 แสดงการกำหนดฟังก์ชันเพื่อแปลงตัว token ให้กับ Little C

วิธีการทำงานของฟังก์ชัน 'get__token ( )' คือการใช้ตัวชี้ (pointer) ชื่อ 'prog' ชี้ไปที่ตำแหน่งของชุดคำสั่งที่กำลังทำการประมวลผลอยู่ในขณะนั้น และตัวแปลภาษาจะไม่ทำการเปลี่ยนค่าของตัวชี้ชื่อ 'p__buf' ที่จะชี้ไปที่ตำแหน่งของจุดเริ่มต้นโปรแกรมที่กำลังทำการประมวลผลอยู่ และเมื่อเริ่มทำการประมวลผลโปรแกรมฟังก์ชัน 'get__token ( )' จะเริ่มทำงานด้วยการอ่านโปรแกรมโดยข้ามส่วนที่เป็นช่องว่างในโปรแกรม (ซึ่งรวมถึงเครื่องหมายจบบรรทัดและเครื่องหมายขึ้นบรรทัดใหม่) และเนื่องจาก token ของภาษาซี (ยกเว้นเครื่องหมายคำพูดหรือตัวอักษรที่เป็นค่าคงที่) จะไม่มีช่องว่างอยู่ด้วย ฉะนั้นช่องว่างทั้งหมดภายในคำสั่งจะไม่มีความหมายใด ๆ กับการเขียนโปรแกรม นอกจากนี้ฟังก์ชัน 'get__token ( )' ก็ยังจะผ่าน (โดยไม่ทำการแปลความหมาย) คำสั่งส่วนที่เป็นคำอธิบายของโปรแกรมด้วย

ต่อจากนั้นถ้าพบว่ามีข้อความที่ใช้มีคุณสมบัติเป็น token จะนำเอาไปเก็บไว้ในตัวแปรชื่อ 'token' และจะนำมาทำการตรวจสอบประเภทของ token (ซึ่งได้กำหนดเอาไว้ด้วยตัวแปรชื่อ 'tok__types') และจะระบุประเภทเอาไว้ในตัวแปรชื่อ 'token__type' แล้วถ้าพบว่า token นั้นเป็นคีย์เวิร์ดจะนำเอาคำสั่งภายในมาเก็บไว้ที่ตัวแปรชื่อ 'tok' โดยเรียกใช้ฟังก์ชัน 'look__up ( )' สำหรับคำสั่งภายในของตัวแปลภาษาของเรานั้นจะได้พูดถึงกันในช่วงต่อไปมาดูกันที่ฟังก์ชัน 'get__token ( )' อีกที ซึ่งจะเห็นได้ว่ามันจะทำหน้าที่แปลงเครื่องหมายเปรียบเทียบให้เป็นอักษรสองตัวตามที่กำหนดไว้ในตัวแปรชื่อ ''doubleopes'' ถึงแม้ว่าวิธีนี้จะไม่ใช่สิ่งจำเป็นในทางเทคนิคแต่จะช่วยให้ parser สามารถทำงานได้สะดวกขึ้น

ในส่วนสุดท้ายของ parser คือส่วนที่จะใช้ตรวจสอบความถูกต้องของกฎเกณฑ์ภาษาซึ่งพบว่าไม่ถูกต้องตามที่วางไว้จะเรียกใช้ฟังก์ชัน 'entx__err ( )' ที่จะใช้แสดงค่าหรือแจ้งให้รู้ถึงลักษณะของความผิดพลาดที่เกิดขึ้น สำหรับฟังก์ชัน 'entx__err ( )' นี้สามารถจะถูกเรียกใช้ได้จากทุก ๆ ส่วนของรูทีนของตัวแปลภาษาของเรา ถ้าพบว่ามีความผิดพลาดเกิดขึ้นที่ส่วนนั้น ๆ ซึ่งโปรแกรมในส่วนของฟังก์ชัน entx__err ( )' ได้แสดงไว้ให้ดูแล้ว

จากชุดคำสั่งต่าง ๆ ในฟังก์ชัน 'entx__err ( )' เห็นได้ว่าสามารถแสดงหมายเลขบรรทัดของโปรแกรมที่เกิดความผิดพลาดขึ้นได้ด้วย (ซึ่งหมายเลขบรรทัดนี้อาจจะเป็นหมายเลขของบรรทัด

ที่ถัดจากบรรทัดที่เกิดความผิดพลาดขึ้นก็ได้) และจะแสดงชุดคำสั่งในบรรทัดที่พบว่ามีความผิดพลาดนั้นให้ยิ่งไปกว่านั้นฟังก์ชัน 'entx__err ( )' จะจบด้วยการเรียกใช้ฟังก์ชัน 'longjump ( )' เสมอ ทั้งนี้เพราะว่าเมื่อมีความผิดพลาดเกิดขึ้นแล้ว (ซึ่งมักจะพบบ่อยในการทำงานกับส่วนของโปรแกรมที่มีการวนลูปหลายระดับหรือการเรียกใช้ตัวเอง) วิธีการที่ง่ายที่สุดในการจัดการกับสถานการณ์ที่เกิดขึ้น คือการกระโดดไปทำงานในจุดที่มีความปลอดภัยมากกว่า ซึ่งการเรียกใช้ฟังก์ชัน 'longjump ( )' จะเป็นการสั่งให้กระโดดไปทำงานที่จุดซึ่งไกลออกไปจากตำแหน่งที่เกิดความผิดพลาดขึ้น ถึงแม้ว่าเป็นสิ่งที่เป็นไปได้ที่จะทำการกำหนดค่าความผิดพลาดเอาไว้ในแฟกซ์ (global error flag) และทำการตรวจสอบค่าของแฟกซ์ในหลาย ๆ จุดของแต่ละโปรแกรมย่อย แต่วิธีการนี้จะทำให้เกิดความล่าช้าในการทำงานของตัวแปลภาษาขึ้นโดยไม่จำเป็น

parser ของ Little C ตามที่เราได้พูดกันไว้แล้วว่า parser ของเราจะใช้เทคนิคแบบ recursive-descent จะเห็นได้ว่าในโปรแกรมจะมีฟังก์ชันต่าง ๆ ที่จำเป็นไว้ให้ใช้ด้วย นอกจากนี้ยังรวมถึงข้อมูลส่วนกลางและประเภทของข้อมูลต่าง ๆ ด้วยโดยจะเก็บโปรแกรมนี้เอาไว้ในไฟล์ชื่อ PARSER.C ซึ่งจะเป็นไฟล์หนึ่งในสามไฟล์ของ Little C

ในโปรแกรม parser นั้นจะมีฟังก์ชันที่เริ่มต้นด้วย 'eval__exp ( )' และฟังก์ชัน 'atom ( )' ซึ่งจะเป็นฟังก์ชันที่ทำหน้าที่ควบคุมกฎที่ใช้สำหรับนิพจน์ต่าง ๆ ของ Little C และเพื่อตรวจสอบความถูกต้องในการทำงานของฟังก์ชันเหล่านี้ เราอาจจำเป็นต้องลองประมวลผล

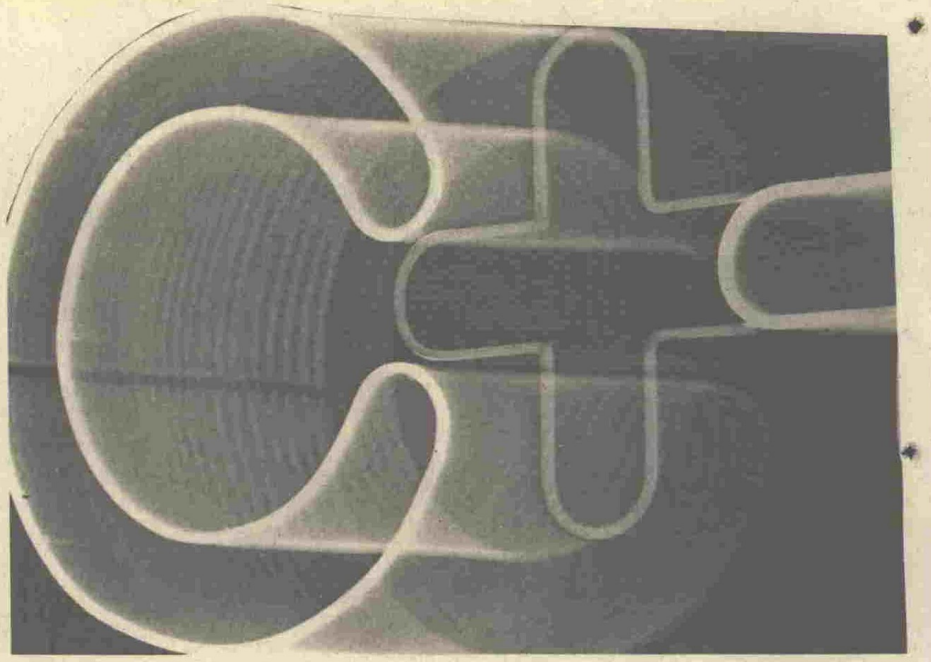
```
/* 'standard library'' functions are declared hear so
that can be put into the internal funcion table that follows
*/
int call__getche (void), call__putch (void):
int call__puts (void), print (void), getnum (void):
struct intern__func__type {
}   char * f__name: /* function name */
    int (*p) ( ): /* pointer to the function */
    intern__func [ ] = {
    ''getche'', call__getche,
    ''putch'', call__putch,
      ''puts'', call__puts,
      ''print'', print,
      ''getnum'', getnum,
      '' '', 0 /* null terminate the list */
};
```

รูปที่ 9 แสดงอะเรย์ชื่อ 'internal-func'

โปรแกรม parser นี้โดยใช้นิพจน์ต่าง ๆ ดู (ถ้าต้องการดูรายละเอียดเกี่ยวกับวิธีการทำงานของนิพจน์ parser ก็สามารถหาดูได้จากหนังสือชื่อ ''C: The Complete Reference 2 nd Edition'' ซึ่งเขียนโดย Herbert Schidt)

สำหรับฟังก์ชัน 'atom ( )' นั้นจะทำหน้าที่หาค่าคงที่หรือตัวแปรรวมถึงฟังก์ชันและค่าคงที่แบบตัวอักษร ส่วนชนิดของฟังก์ชันนั้นมีอยู่ด้วยกันสองแบบที่สามารถใช้ในโปรแกรมได้ คือแบบที่ผู้ใช้กำหนดขึ้นเองและที่กำหนดไว้แล้วในตัวแปลภาษาซึ่งถ้า Little C พบว่าเป็นฟังก์ชันที่ผู้ใช้สร้างขึ้นเองตัวแปลภาษาของเราก็จะทำการประมวลผลเพื่อหาค่าของฟังก์ชันนั้น อย่างไรก็ตามถ้าพบว่าฟังก์ชันนั้นมีอยู่ในตัวแปรภาษาก็จะทำการเรียกใช้ฟังก์ชัน 'internal__func ( )' เพื่อหาตำแหน่งของฟังก์ชันแล้วจึงเรียกใช้ฟังก์ชันนั้นต่อไปตำแหน่งที่อยู่ของฟังก์ชันจะถูกเก็บเอาไว้ในอะเรย์ชื่อ 'internal__func' ด้วยตามที่จะได้แสดงให้เห็นดังรูปที่ 9

ตามโปรแกรมของเราจะเห็นได้ว่า Little C นั้นมีฟังก์ชันภายในไว้ให้เพียงไม่กี่ชุดแต่ถ้าเราต้องการที่จะเพิ่มเติมฟังก์ชันต่าง ๆ ที่ต้องการสามารถทำได้



ด้วยวิธีการอย่างเดียวกันกับที่แสดงไว้ในโปรแกรม ซึ่งจะเห็นว่าทำได้อย่างสะดวกสบาย สำหรับฟังก์ชันที่จะต้องเรียกใช้ต่อจากฟังก์ชันต่าง ๆ ที่กำหนดไว้นี้จะถูกเก็บเอาไว้ในไฟล์อื่นแยกต่างหาก ซึ่งเราจะได้พูดถึงไฟล์นั้นในโอกาสต่อไป

จุดสำคัญอีกอย่างหนึ่งเกี่ยวกับโปรแกรมย่อยต่าง ๆ ที่มีอยู่ใน expression parser คือทำการตรวจสอบความถูกต้องของโปรแกรมต้นฉบับของภาษาซีที่จะนำมาเขียนโปรแกรมกับตัวแปลตัวนี้ ซึ่งในการทำงานนั้นโปรแกรมเหล่านี้จำเป็นต้องใช้วิธีการค้นหา token ตัวที่อยู่ถัดไป (token lookahead) ด้วย ตัวอย่างเช่น เพื่อให้ Little C รู้จักว่า 'count' ที่อยู่ในฟังก์ชันนั้นทำหน้าที่เป็นฟังก์ชันโดยไม่ใช่ตัวแปรนั้นก็จะต้องทำการอ่านคำว่า 'count' และเครื่องหมายวงเล็บที่อยู่ต่อจากคำว่า 'count' นั้นด้วย ตัวอย่างเช่น

```
alpha = count ( );
```

คำสั่งข้างต้นเมื่อ parser พบว่ามีเครื่องหมายวงเล็บอยู่ต่อจากคำว่า "count" ก็จะทำให้รู้ว่าสิ่งที่พบนั้นเป็นฟังก์ชันไม่ใช่ตัวแปร ในทำนองเดียวกันเมื่อพบว่า "count" ไม่มีเครื่องหมายวงเล็บต่อท้ายจะรู้ได้ทันทีว่าเป็นตัวแปร เช่น

```
alpha = count *10;
```

เมื่อตรวจพบว่าหลังคำว่า "count" เป็นเครื่องหมายคูณ ('*') ก็จะรู้ว่าเป็นชื่อตัวแปร ฉะนั้นจะทำการส่งค่า 10 เข้าไปเก็บไว้ในส่วนของข้อมูล (input stream) และด้วยเหตุผลนี้จึงทำให้ expression parser จำเป็นต้องเรียกใช้ฟังก์ชัน 'putback ( )' เพื่อส่งค่าของ token ตัวสุดท้ายไปเก็บไว้ในส่วนของข้อมูลของโปรแกรมนอกเหนือจากฟังก์ชันที่ได้พูดถึงกันไปแล้ว ใน expression perser ของเราก็ยังมีฟังก์ชันต่าง ๆ อีกหลายชุดที่ไม่ได้หยิบยกเอามาพูดกัน แต่อย่างไรก็ตามเมื่อเราได้ใช้งาน parser ตัวนี้ดูแล้วจะทำให้เข้าใจถึงวิธีการทำงานของฟังก์ชันต่าง ๆ เหล่านั้นได้ดีในโอกาสต่อไป

## ก่อนจากกัน

ในตอนที่ 1 เป็นเรื่องราวทั่ว ๆ ไปของตัวแปลภาษาชื่อ Little C ได้อธิบายถึงหลักเกณฑ์ โครงสร้างของตัวแปลภาษา วิธีการทำงานและการเขียนโปรแกรม parser ซึ่งถือว่าได้ว่าเป็นหัวใจของ Little C ไปเรียบร้อยแล้ว ในตอนหน้าเราก็จะได้มาพูดกันถึง Little C ในแบบเต็มรูป และก็จะหมายถึงการเขียนโปรแกรม Little C ทั้งชุด ซึ่งจะทำให้ได้ตัวแปลภาษาไว้สร้างความเพลิดเพลินได้อีกหนึ่งชุด นอกเหนือจากที่เรียนรู้ถึงวิธีการใช้ภาษาซีเพื่อสร้างตัวแปลภาษา ถ้าเราศึกษาวิธีการเขียนโปรแกรมภาษาซีจนเข้าใจอย่างถ่องแท้แล้ว จะสามารถทำให้นำไปเขียนโปรแกรมเพื่อทำงานอย่างอื่นได้ เพราะโปรแกรมที่ใช้ทำงานในลักษณะนี้จัดว่าเป็นโปรแกรมประเภทควบคุมระบบ ซึ่งต้องการความรู้ความชำนาญเกี่ยวกับคอมพิวเตอร์และภาษาซีมากพอควรเลยทีเดียว

ถ้าสนใจเรื่องราวการสร้าง Little C สามารถติดตามตอนหน้าได้ครับ สำหรับตัวโปรแกรม Little C ทางกองบรรณาธิการได้จัดทำเป็นซอฟต์เมล์ไว้บริการแล้วครับ

### เอกสารอ้างอิง

Herbert Schildt, "Born to Code in C," Osborne McGraw-Hill, Inc., 1989.